RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

IN-61
43092
P.15

Mutual Exclusion

Peter J. Denning

21 Mar 91

RIACS Technical Report TR-91.13

NASA Cooperative Agreement Number NCC 2-387

Mutual Exclusion

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-91.13
21 Mar 91

Almost all computers today operate as part of a network, where they assist people in coordinating actions. Sometimes what appears to be a single computer is actually a network of cooperating computers; for example, some supercomputers consist of many processors operating in parallel and exchanging synchronization signals. One of the most fundamental requirements in all these systems is that certain operations be indivisible: the steps of one must not be interleaved with the steps of another. Two approaches have been designed to implement this requirement, one based on central locks and the other on distributed ordered tickets. Practicing scientists and engineers need to come to become familiar with these methods.

This is a preprint of the column *The Science of Computing* for *American Scientist* 79, No. 2 (March-April 1991).

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

Mutual Exclusion

Peter J. Denning

Research Institute for Advanced Computer Science

21 Mar 91

Almost all computers today operate as part of a network. They assist people in coordinating actions, such as sending or receiving electronic mail, depositing or withdrawing funds from bank accounts, and making or cancelling reservations. Sometimes what appears to be a single computer is actually a network of cooperating computers. For example, certain supercomputers consist of many processors operating in parallel; each of the component processors starts and stops various tasks in accordance with signals it exchanges with other processors.

One of the most fundamental requirements in all of these systems is that certain operations be indivisible: the operations must be carried out in some definite order, one at a time, even when different computers request them simultaneously. If, contrary to this requirement, the instructions of one operation were interleaved with those of another, the results would be unpredictable. Deposits and withdrawals could be lost; confirmed reservations might disappear; parallel-processing computers could produce invalid outputs.

Because indivisible operations are not allowed to be in progress simultaneously, we say that they are mutually exclusive. Until recently, methods for guaranteeing mutual exclusion have been little known outside the community of computer scientists working on operating systems and networks. But now shared databases and parallel supercomputers make it necessary for practicing scientists and engineers to come to grips with mutual exclusion. This essay is devoted to the subject.

The earliest formulation of the problem of mutual exclusion is credited to Edsger W. Dijkstra, then at the Technological University of Eindhoven in the Netherlands *(1)*. He considered a model of a typical configuration of a multicomputer: processors numbered 1 through N have access via a network to a shared resource R that they read and update *(Figure 1)*. The shared resource might be the accounting records of a bank or the reservation records of a hotel or airline. The individual computers run programs that read and alter the shared resource--for example, programs that make deposits and withdrawals from bank accounts or programs that make and cancel reservations. If these programs can run simultaneously, they can leave the shared resource in an invalid state, causing considerable damage or inconvenience to individuals and organizations. Dijkstra proposed that the procedures that manipulate the shared resource be organized as a set of indivisible operations. He used the term mutual exclusion to refer to the property that the steps of one operation cannot be interleaved with the steps of another. He referred to any sequence of instructions that must not be interleaved with other sequences of instructions as a critical section.

Dijkstra specified that a valid solution to this problem must satisfy three properties. First, it must be symmetric: The same protocol must work on any processor. Second, no assumptions can be made about the relative speeds of the processors; they can speed up, slow down, and overtake one another in unpredictable ways. Third, the failure of any processor outside a critical section must not affect the future operation of the other processors.

Noting that the processors must exchange signals that allow them to determine which one may next enter its critical section, Dijkstra proposed to store the variables used for these signals in a memory unit accessible to all processors. The operations of reading and writing memory locations in this common store must themselves be indivisible--that is, when two or more computers try to communicate simultaneously with the same common location, these communications will take place one after the other, although in an unknown sequence. This assumption about memory reads and writes is satisfied by common memory architectures *(Figure 2)*.

The essence of Dijkstra's solution is to attach a standard prologue and epilogue to the code of a critical section *(Figure 3)*. The prologue protocol seeks permission for access to the shared resource and obtains it after a delay of unknown duration; the epilogue signals the other waiting computers that the shared resource is now free. A simple example exists for computers that recognize a test-and-reset instruction: $TR(a)$ returns the contents of memory location a and replaces those contents with 0, all in one indivisible operation. The binary variable k, initially 1, is used to indicate when the resource is free ($k=1$) or in use ($k=0$). The solution is:

```
L:   if $TR(k)=0$ then goto $L$
     indivisible operation on resource $R$
     $k=1$
```

The computer will loop at statement L as long as $k=0$; as soon as the resource is released, k becomes 1 and the next computer that tries statement L will pass, leaving $k=0$. Mutual exclusion of memory operations guarantees that no more than one processor can pass statement L after k becomes 1.

This solution is easy to understand but has four drawbacks. First, waiting processors loop at statement L; not only are those processors unavailable for other useful work in the interim, but their recurring contention for access to the shared memory area slows down the other computers. This situation is called busy waiting. Second, a failure of the memory holding k will halt the system. Third, there is no guarantee that the computer that has been waiting the longest time will be next to use the resource; any waiting computer can go next. Fourth, this solution is not available on a system that has no test-and-reset instruction. How can these drawbacks be eliminated?

Dijkstra's algorithm *(Figure 4)* overcomes the fourth drawback by relying only on normal read and write operations. It is a difficult program to understand because one must visualize multiple processors proceeding simultaneously through the code at different speeds and in different orders.

Donald Knuth of Stanford University provided an even more complicated program that overcomes the third drawback *(2,3)*. It ensures that all processors take turns in executing their critical sections.

Overcoming the second drawback requires a different strategy that avoids storing a lock in a single, common location. Leslie Lamport of the Digital Equipment Corporation proposed a strategy he called the "Baker's algorithm" because it was modeled on the method of serving customers in the order of numbers they draw from a ticket machine on entering a bakery *(4)*. Lamport did not want to posit an electronic ticket dispenser because the failure of such a device would halt the system. Instead, he proposed that each processor compute its own ticket number as one larger than the maximum of all the numbers it sees held by the other processors. His protocol needs some additional checks in it to deal with the case where two computers compute the same ticket number simultaneously *(Figure 5)*.

Even though ticket dispensers may be no less fault-tolerant than locks stored in shared memory, David Reed and Rajendra Kanodia demonstrated that a system with such devices can perform many useful synchronization operations, including mutual exclusion *(5)*. Let the variable k denote the number of completions of a critical section (k=0 when the system is started). The solution is as simple as the previous one, where *ticket* is a call on the function that returns the next ticket from the ticket dispenser:

```
    t=ticket
L:  if t > k then goto L
    indivisible operation on resource R
    k = k + 1
```

Like the previous solution, this one also suffers from the problem of busy waiting at statement L.

Busy waiting is a persistent problem in both the central-lock approach and the ticket approach: there is a loop in which processors can cycle while waiting their turn for the shared resource. Busy waiting consumes processor time, which may be a waste if there is other useful work to do. Busy waiting also slows down the nonwaiting processors because the repeated tests by the waiting processors consume memory cycles that would otherwise be available. A high degree of busy waiting, therefore, affects the whole system. There are two ways to mitigate busy waiting. One is to insert a pause before looping back; the duration of the pause should be approximately $KT/2$, where K is the average number of computers waiting for the shared resource and T is the average duration of the critical operation. This reduces the frequency of testing to approximately the frequency of completions of the critical operations.

The other method is to devise a system of queues on each computer that keeps track of the different computational processes on that computer and allows a process that encounters a lock to be suspended and the processor reassigned to another process. Such a method was proposed by Dijkstra in 1968 *(6)* and has undergone many refinements since then *(7)*. The essence of Dijkstra's approach is a new programming object called a semaphore, used for signalling among processes. A semaphore consists of a counter and a queue of processes awaiting a signal. In addition to the queues associated with semaphores, the operating system maintains a queue called the ready list of processes waiting their turn for execution on a processor. Dijkstra proposed two indivisible operations on semaphores:

```
wait(s):
 count(s) = count(s) - 1
 if s < 0 then
  add self to queue(s)
  switch processor to process at head of ready list
 end if
```

```
signal($s$):
 if $s < 0$ then
   transfer process from head of $queue(s)$
       to tail of ready list
 end if
 $count(s) = count(s) + 1$
```

A process executing a wait operation may encounter an unknown delay, and the processor will be reassigned to a ready process in the meantime. A process executing a signal operation always completes that operation immediately and may awaken one of the other processes waiting in the queue; the latter process will resume and complete its previously interrupted wait operation.

With a semaphore s whose initial count is 1, the critical section can be implemented as

```
wait($s$)
indivisible operation on resource $R$
signal($s$)
```

This semaphore-based solution only avoids busy waiting in an operating system that multiplexes a set of processors among a set of ready processes--the mode of operation usually called time-sharing. Even then, some busy waiting may be encountered during queue manipulation *(7)*. If a computer runs a single dedicated process, there is no way to avoid busy waiting when it encounters a busy shared resource.

It is also important to note that some form of unknown delay must always exist when waiting for access to a shared resource when simultaneous requests must be arbitrated. Even the lowly arbiter within the memory subsystem cannot have a fixed time limit placed on when it will make its selection among nearly simultaneous requests *(8)*.

The main conclusion is that we have a variety solutions to the problem of mutual exclusion in systems of concurrent programs. The solutions are difficult to get right. Even the most experienced programmers make mistakes with them.

References

1. Edsger Dijkstra. 1965. "Solution of a problem in concurrent programming control." *Communications of ACM 8*, 9 (September), 569.
2. Donald Knuth. 1966. "Additional comments on a problem in concurrent programming control." *Communications of ACM 9*, 5 (May), 321-322.
3. Edward Coffman and Peter Denning. 1973. *Operating Systems Theory*. Prentice-Hall. See Chapter 2.
4. Leslie Lamport. 1974. "A new solution of Dijkstra's concurrent programming problem." *Communications of ACM 17*, 8 (August), 453.455.
5. David Reed and Rajendra Kanodia. 1980. "Synchronization with eventcounters and sequencers." *Communications of ACM 22*, 2 (February), 115-123.
6. Edsger Dijkstra. 1968. "The structure of THE multiprogramming system." *Communications of ACM 11*, 5 (May 1968), 341-346.
7. Peter Denning, T. Don Dennis, and Jeffrey Brumfield. 1981. "Low contention semaphores and ready lists." *Communications of ACM 24*, 10 (October), 687-698.
8. Peter Denning. 1985. "The arbitration problem." *American Scientist 73*, 6 (November-December), 516-518.

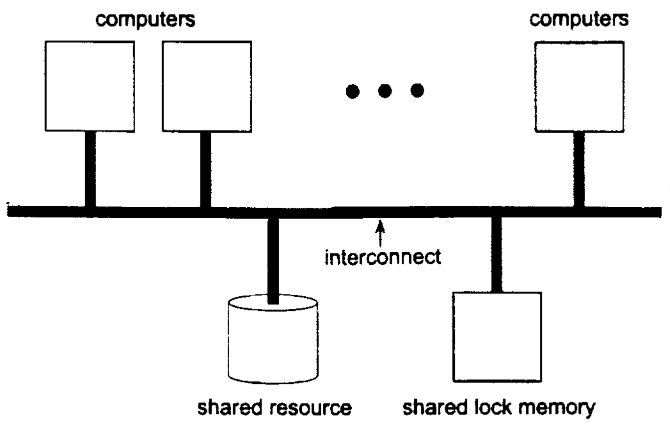

Figure 1. Mutual exclusion problem arises in any network of computers in which multiple processors must gain access to a shared resource, such as a database. Here computers 1 through N require access to resource R. If two machines attempted to use R at the same time, the information in R could be left in an inconsistent state; to eliminate this possibility, only one computer at a time can be allowed access to R. Some of the simplest and earliest schemes for ensuring mutual exclusion make use of "lock variables," k, held in a region of memory available to all the computers.

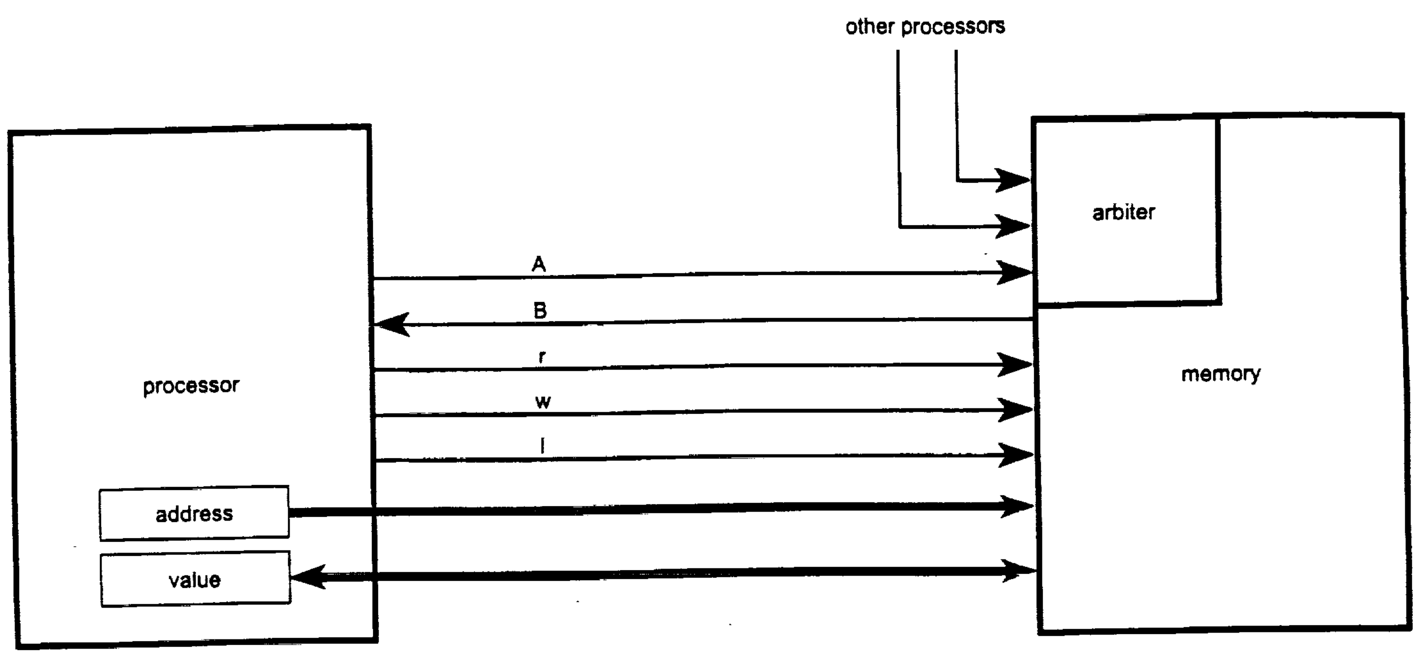

Figure 2. Interface between processors and memory has an important role in guaranteeing mutual exclusion. The interface shown here has five control lines and two data paths. The processor can read from the memory, write to it, or perform a test-and-reset operation, which reads from a location and writes a value of zero into it in a single, atomic action. For each operation the processor first places the address of the selected memory location in the address register and then sets the control line *A* to one to request access to memory. An arbiter, which resolves competing requests for access, replies on line *B* when the processor is granted access. Now the processor issues a signal on either the *r*, the *w* or the *l* line to select an operation. In response to an *r* signal, the value at the addressed location is read into the value register. A *w* signal causes the contents of the value register to be written into the addressed location. Finally the *l* line calls for a test-and-reset operation, reading the contents of a location into the value register and resetting the location to zero. On completion, the processor sets the *A* line to zero to indicate it is done.

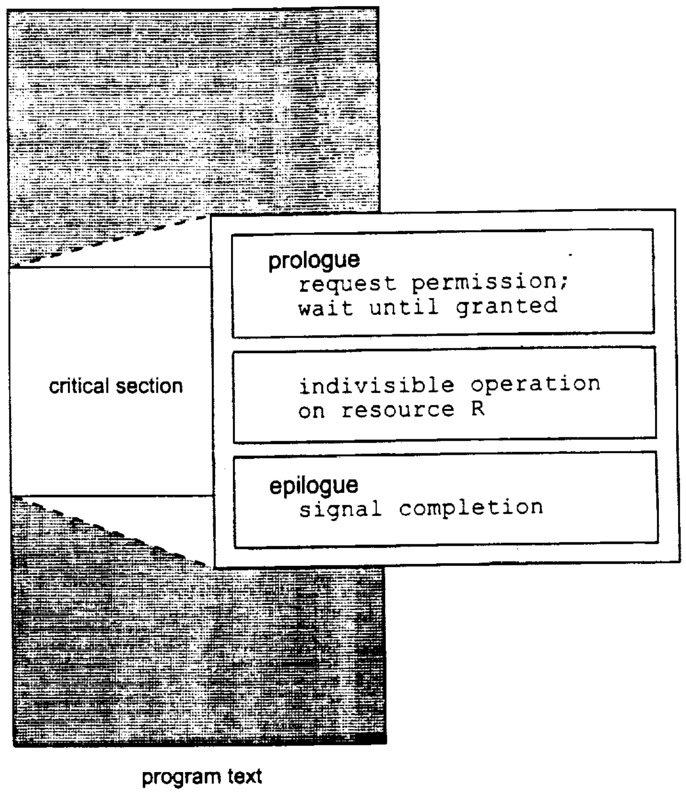

Figure 3. Critical section of a computer program, requiring access to a shared resource *R*, must be executed as an indivisible unit, without interruption by other concurrent programs that also access *R*. In a standard protocol for ensuring exclusive access, a prologue does not allow the indivisible operation to begin until permission is granted; an epilogue signals completion of the operation, so that other processors can take their turn.

prologue

```
    b[i] = 0
L:  if k ≠ i
      then
        c[i] = 1
        if b[k] = 1 then k = i
        goto L
      else
        c[i] = 0
        for j = 1 step 1 until n do
          if j ≠ i and c[j] = 0 then goto L
    end if
```

epilogue

```
    b[i] = 1
    c[i] = 1
```

Figure 4. Edsger W. Dijkstra's algorithm for mutual exclusion employs a central lock as a kind of turnstyle, allowing just one processor at a time to reach the resource *R*. The variable *k*, stored in a location available to all the processors, serves as the lock. The variables *b* [*i*] and *c* [*i*], held in the local memory of processor *i*, together convey the status of each process; *b* [*i*] is zero when process *i* is in its critical section, and *c* [*i*] is zero if *i* has tentative permission to use resource *R*. In the prologue, process *i* seeks to set *k* to *i*; if it succeeds, it has obtained tentative permission. The permission is only tentative because another process may have seized the chance to set *k* simultaneously. The statements of the else clause eliminate this possibility by explicitly checking the *c* variables of all the other processes. The prologue algorithm is difficult to understand because it must operate correctly when many processors execute it at various speeds and in various sequences.

prologue

```
     c[i] = 1
     n[i] = 1 + max(n[1],...,n[N])
     c[i] = 0
     for j = 1 step 1 until n do
L1:     if c[j] ≠ 0 then goto L1
L2:     if n[j] ≠ 0 and (n[j],j) < (n[i],i)
        then goto L2
     end for
```

epilogue

```
    n[i] = 0
```

Figure 5. Leslie Lamport's algorithm is modeled on the practice of issuing sequentially numbered tickets to customers at a bakery. To ask permission, processor *i* takes a number that is 1 larger than the largest of all the numbers previously taken; it stores the number in variable *n* [*i*] in its local memory. The comparison performed in the statement labeled *L* 2 forces processor *i* to wait until its number is the smallest of those held by all the waiting processors. In case two processors took a number at the same moment (and therefore calculated the same value), the indexes *i* and *j* are also compared; the processor with the lower index is allowed to proceed first. Statement *L* 1 prevents a subtle bug: If processor *j* takes a number at the same time as processor *i* but does not store that number in *n* [*j*] until processor *i* has passed the test in *L* 2, both processors might be granted access to *R*. Statement *L* 1 delays the test until processor *j* has stored its ticket number.